CPSC 311: **Practice** Midterm Exam #1

October 2, 2018

- Instructions true for both the practice exam and actual exam:
  - **ALL PROBLEMS** refer to the `ElastiPerl` language and parser/interpreter attached. (This is identical to the code handout for the tutorial except with less critical sections of the code—including most tests and error handling—removed for space.)
  - **EACH PROBLEM** explores a change to the given language. The change discussed in a particular problem should be considered for **that problem alone**. Each problem starts "fresh" from the provided language and parser/interpreter.

Instructions for the practice exam alone:

We anticipate asking questions similar in structure or spirit on the actual exam. Students who have carefully worked through and understood these problems will be at an **immense** advantage in the exam over those who have not. In each question, we include at least one **PRACTICE NOTE** to describe how the actual exam may relate to this practice exam. Note also that while the problem titles are not meaningful, they will match between actual exam questions and related practice exam questions. (On the actual exam, we include brief **EXAM NOTES** to remind you how the exam problems correspond with practice problems.)

Note that we may not ask all of these questions, or we may add questions, but at this time this practice exam does explore the full set of questions under consideration.

We did *not* scale the practice exam to be of an appropriate length for 50 minutes.

1 I'm Syntax, Your Nemesis [?? marks]

PRACTICE NOTE: *We anticipate asking a similar but more closed-ended question on the exam.*

Here is the original syntax for the EP language, expressed in EBNF:

```
<Expr> ::= <num> | <id>

         | {<binop> <Expr> <Expr>}

         | {fun {<id>} <Expr>}

         | {with {<binding>*} <Expr>}

         | {extend <Expr> <mapping>}

         | {<Expr> <Expr>}              ; function application


<binop> ::= + | - | * | /


<binding> ::= {<id> <Expr>}

<mapping> ::= {<Expr> -> <Expr>}
```

1. Neatly edit the EBNF above to add a new way to bind identifiers inside of a `with` expression in order to conveniently define a function. **[?? marks]**

   This should look like:

   ```
   {with {{def {double n} {+ n n}}}
     {double 5}}
   ```

   which would be equivalent to:

   ```
   {with {{double {fun {n} {+ n n}}}}
     {double 5}}
   ```

   Note that it should be possible to freely mix original- and function-style bindings in a single `with` expression, e.g.:

   ```
   {with {{x 5}
          {def {add-x y} {+ x y}}}
     {add-x 10}}
   ```

2. Select the easier strategy of (1) implementing this as a desugaring or (2) introducing a new `withfun` AST node and implementing this new binding type in interpretation, and then complete the implementation. Argue for why your strategy is easier and for whether it is *better*. **[?? marks]**

2 Dynaguise [?? marks]

PRACTICE NOTE: *We anticipate asking a question with nearly the same structure on the exam.*

An `extendClosureV` is a variant of `Value` defined as:

```
[extendClosureV (key Value?) (rexp Expr?) (env Env?) (enclosed-fun Value?)]
```

It stores the unevaluated result expression (`rexp`) and the environment (`env`) in which the `extend` expression was evaluated. We use this code to apply an `extendClosureV`:

```
[extendClosureV (kval rexp cenv fval)                               ;     if the argument
  (if (and (numV? kval) (numV? aval) (= (numV-n kval) (numV-n aval))) ; <-- matches the key
    (helper rexp cenv)      ; <-- then evaluate the result in the extendClosureV's environment
    (apply fval aval))]))   ; <-- else, apply the contained function value to the argument
```

PRACTICE NOTE: *The actual exam will include both approaches below and add a third approach.*

Consider **two** approaches to that result expression:

1. The **original** approach above.
2. The **body only** approach: We delete the environment from the `extendClosureV` variant and use the current environment of the expression when applying an `extendClosureV` value instead.

Now, answer the questions below.

1. Neatly edit the value type and interpreter above as needed to implement the **body only** strategy.

   PRACTICE NOTE: *The actual exam will request an implementation but not other changes to the broader interpreter. (Any broader changes will either be assumed to be made or be unnecessary.)*

   In addition, clearly explain the one other change (or set of changes) required in the broader interpreter to implement these semantics. **[?? marks]**

2. Does the **body only** approach introduce dynamic scoping? Fill in the circle next to the **best** answer. **[?? marks]**

   ◯ Yes, because the environment used is not the static environment of the `extend` expression.
   ◯ Yes, because the parameter leaks dynamic scope into the result expression.
   ◯ No, because mappings cannot directly access the extended function's parameter.
   ◯ No, because dynamic scoping requires function **definition** not just function application.
   ◯ It's not possible to tell with the details given in this problem.

   PRACTICE NOTE: *The actual exam's multiple choice questions will have similar style but different content.*

3. Each of the following programs in EP has one or two boxes after it for you to fill in the value of the program. Each box is labelled with semantics to use in evaluation (**original** or **body only**). Write in each box the value of the program under the requested semantics. **[?? marks]**

   PRACTICE NOTE: *The actual exam version of this question will have identical instructions* **except** *it will have one to* **three** *boxes with a third approach introduced.*

   NOTE: Write one of three things in the boxes:

   - If the program evaluates to a number, write the number it evaluates to. E.g., for the program `{+ 1 2}`, write `3`.
   - If evaluation of the program runs forever or stops with an error, write the word `error`. You should **not** write the error itself. E.g., for the program `{4 1}`, write `error`.

- If the program evaluates to anything besides these (including closures of either type), write the word `other` in the box. You should **not** write the value itself. E.g., for the program `{fun {x} 1}`, write `other`.

You **can and should** rely on the indentation being reasonable as a guide to the structure of the programs: **[?? marks per blank]**

(a) `{+ 1 4}`

Body only:

(b) `{fun {x} y}`

Original:

(c)
```
{{extend {fun {x} {+ x 10}}
         {y => 100}}
 {with {{y 3}} y}}
```

Original: **Body only:**

(d)
```
{with {{x 100}
       {f {extend {fun {x} {+ x 10}}
                  {x => y}}}}
  {with {{x 1} {y 1000}}
    {f 1}}}
```

Original: **Body only:**

(e)
```
{with {{x 100}
       {f {extend {fun {x} {+ x 10}}
                  {x => y}}}}
  {with {{x 1} {y 1000}}
    {f 100}}}
```

Original: **Body only:**

(f)
```
{{extend {fun {x} 100}
         {{fun {y} y} => 10}}
 {fun {z} z}}
```

Original:

(g)
```
{with {{g {with {{f {fun {x} {+ x 10}}}}
            {extend f {{f 1} => {f 10}}}}}}
  {g {g 1}}}
```

Original: **Body only:**

(h)
```
{with {{f {fun {x} {+ x 10}}}
       {g {extend f {{f 1} => {g 10}}}}}
  {g {g 1}}}
```

Body only: $\boxed{}$

(i)
```
{with {{x 0}
       {f {extend {fun {x} x}
                  {1 => {/ 1 x}}}}
       {x 1}}
  {f 2}}
```

Original: $\boxed{}$ **Body only:** $\boxed{}$

(j)
```
{with {{x 0}
       {f {extend {fun {x} x}
                  {1 => {/ 1 x}}}}
       {x 1}}
  {f 1}}
```

Original: $\boxed{}$ **Body only:** $\boxed{}$

3 The Undefiner [?? marks]

PRACTICE NOTE: *We anticipate asking a question on the exam in exactly the same style as this one but with different extensions to the language.*

We want to add three new forms to our language:

```
<Expr> ::= {undefine <id> <Expr>}
         | {unmap <Expr> {<Expr> => ..}}
         | {withdef <id> <Expr>}
```

`Value` has been edited to add the variant `[undefV]`. Evaluating a `{undefine <id> <Expr>}` form causes `<id>` to be bound to `(undefV)` with a static scope of the `<Expr>`, and then returns the result of evaluating the `<Expr>`. To evaluate a `{unmap <Expr> {<Expr> => ..}}` form, we evaluate the first `<Expr>` to a function value f (a `closureV` or `extendClosureV`), evaluate the second `<Expr>` to a numeric value n, and then produce a function value that extends f such that if it is applied to n, it produces `(undefV)` and otherwise it applies f to the argument value and returns the result. A `withdef` evaluates to `(undefV)` if `<id>` is bound to `(undefV)` and otherwise evaluates `<Expr>` and returns its value. (It produces an error if `<id>` is unbound.)

1. Here are the parser cases for `extend` and a new (correct) one for `undefine`. **Neatly add** a new parser case for `{unmap <Expr> {<Expr> => ..}}` that is implemented by desugaring it using the new `{undefine <id> <Expr>}` form and the previously existing ones. (Note that `..` is just syntax constisting of two consecutive periods and not some special EBNF symbol.) **[?? marks]**

```
(define (parse sexp)
  (local [(define binops ...)]
    (match sexp ...
      [(list 'extend fun-expr (list key-expr '=> res-expr))
       (extend (parse fun-expr) (parse key-expr) (parse res-expr))]
      [(list 'undefine (? valid-id? id) body-expr)
       (undefine id (parse body-expr))]



      [_ (error 'parse "Something went wrong in ~a" sexp)])))
```

2. Here is the existing datatype used for the language's abstract syntax. **Neatly complete** the new variants named `undefine` and `withdef` to represent the `undefine` and `withdef` concrete syntax. **[?? marks]**

```
(define-type Expr
  [num (n number?)]
  [id (name symbol?)]
  [binop (op procedure?) (lhs Expr?) (rhs Expr?)]
  [fun (param symbol?) (body Expr?)]
  [extend (fun-exp Expr?) (key-exp Expr?) (result-exp Expr?)]
  [app (fun-exp Expr?) (arg-exp Expr?)]

  [undefine

  [withdef
```

3. Neatly edit the interpreter to implement the desired behaviour for `undefine` and `withdef`. **[?? marks]**

```
(define (interp-env exp env)
  (local [(define (apply fval aval)
            (type-case Value fval
              [numV (n) (error 'interp-env "attempt to apply a number ~a as a function in ~a"
                               n exp)]
              [closureV (param body cenv)
                        (helper body (anEnv param aval cenv))]
              [extendClosureV (kval rexp cenv fval) ; <-- shadowing fval, which is OK
                              (if (and (numV? aval) (= (numV-n kval) (numV-n aval)))
                                  (helper rexp cenv)
                                  (apply fval aval))]))

          ...

          (define (helper exp env)
            (type-case Expr exp
              ...
              ; Insert your cases here.
              [undefine (               )



                  ]
              [withdef (               )



                  ]

              [extend (fexp kexp rexp)
                      (extendClosureV
                       (assert-type (helper kexp env) numV?) rexp env
                       (assert-type (helper fexp env) (or/c closureV? extendClosureV?)))]
              [app (fexp aexp)
                   (apply (helper fexp env) (helper aexp env))]))]
    (helper exp env)))
```

4 Mint Condition in its Original Set-Box [?? marks]

PRACTICE NOTE: *We anticipate asking precisely the same question on the exam except for the particular programs whose values we request.*

Imagine we have correctly added mutation via boxes to our language, with these extra syntactic forms using their usual semantics (where `set-box!` returns the new value placed in the box):

```
<Expr> ::= {box <Expr>}
         | {set-box! <Expr> <Expr>}
         | {unbox <Expr>}
```

This requires defining an order of evaluation of subexpressions. Where that is not already explicit in the language semantics, it will be left-to-right within an expression.

In each box after a program below, write the result of evaluating the program as a number (e.g., for the program `{+ 1 2}`, write 3). **All results** are numbers. **[?? marks per problem]**

1. `{unbox {box 3}}`

2.
```
{with {{b {box 1}}
       {f {fun {x} {set-box! x {* 10 {unbox x}}}}}
       {g {extend f {1 => {set-box! b {* 2 {unbox b}}}}}}}
  {+ {g b}
     {g 1}}}
```

3.
```
{with {{f  {fun {x} {box 1}}}
       {b1 {f 1}}
       {b2 {f 1}}}
  {+ {set-box! b1 {* 10 {unbox b1}}}
     {set-box! b2 {* 10 {unbox b2}}}}}
```

4.
```
{with {{f  {fun {x} {box 1}}}
       {g  {extend f {1 => {box 1}}}}
       {b1 {g 1}}
       {b2 {g 1}}}
  {+ {set-box! b1 {* 10 {unbox b1}}}
     {set-box! b2 {* 10 {unbox b2}}}}}
```

5.
```
{with {{f {fun {x} x}}
       {g {extend f {2 => {box 10}}}}
       {b {box 1}}}
  {+ {set-box! b 2}
     {unbox {g b}}}}
```

5 Bonus [?? BONUS points]

PRACTICE NOTE: There may be bonus question(s) on the exam. If there are, you should heed this warning we'll post at the top of the bonus questions:

Bonus marks add to your exam and course bonus mark total but are **not** required. **WARNING**: These questions are too hard for their point values. We are free to mark these questions harshly. Finish the rest of the exam before attempting these questions. Do not **taunt** these questions.

This page intentionally left (almost) blank.
If you write answers here, you must CLEARLY indicate on this page what question they belong with AND on the problem's page that you have answers here.